Introduction to GNU Radio Companion

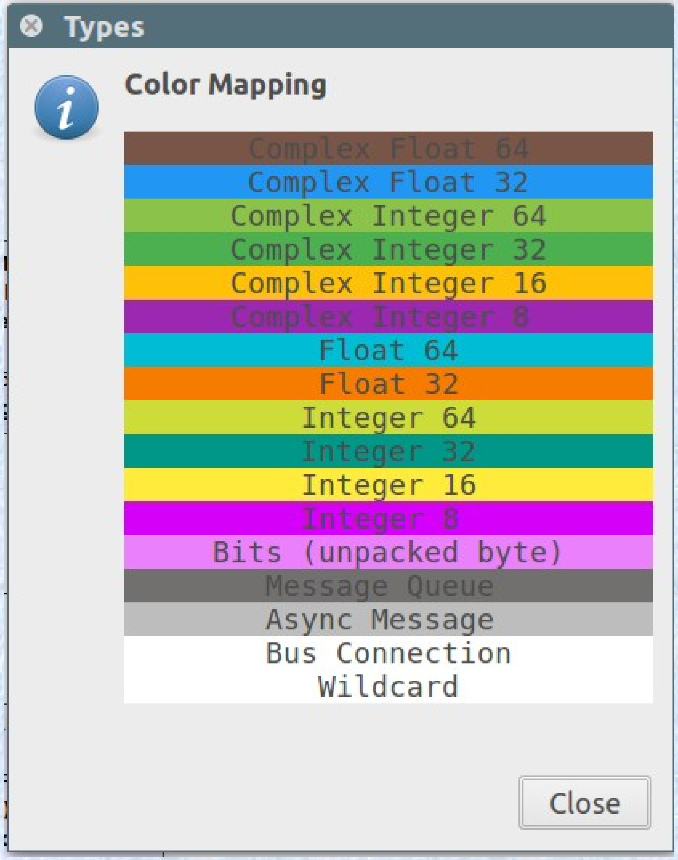
NBFM, Python Blocks, Community

Derek Kozel - MW0LNA

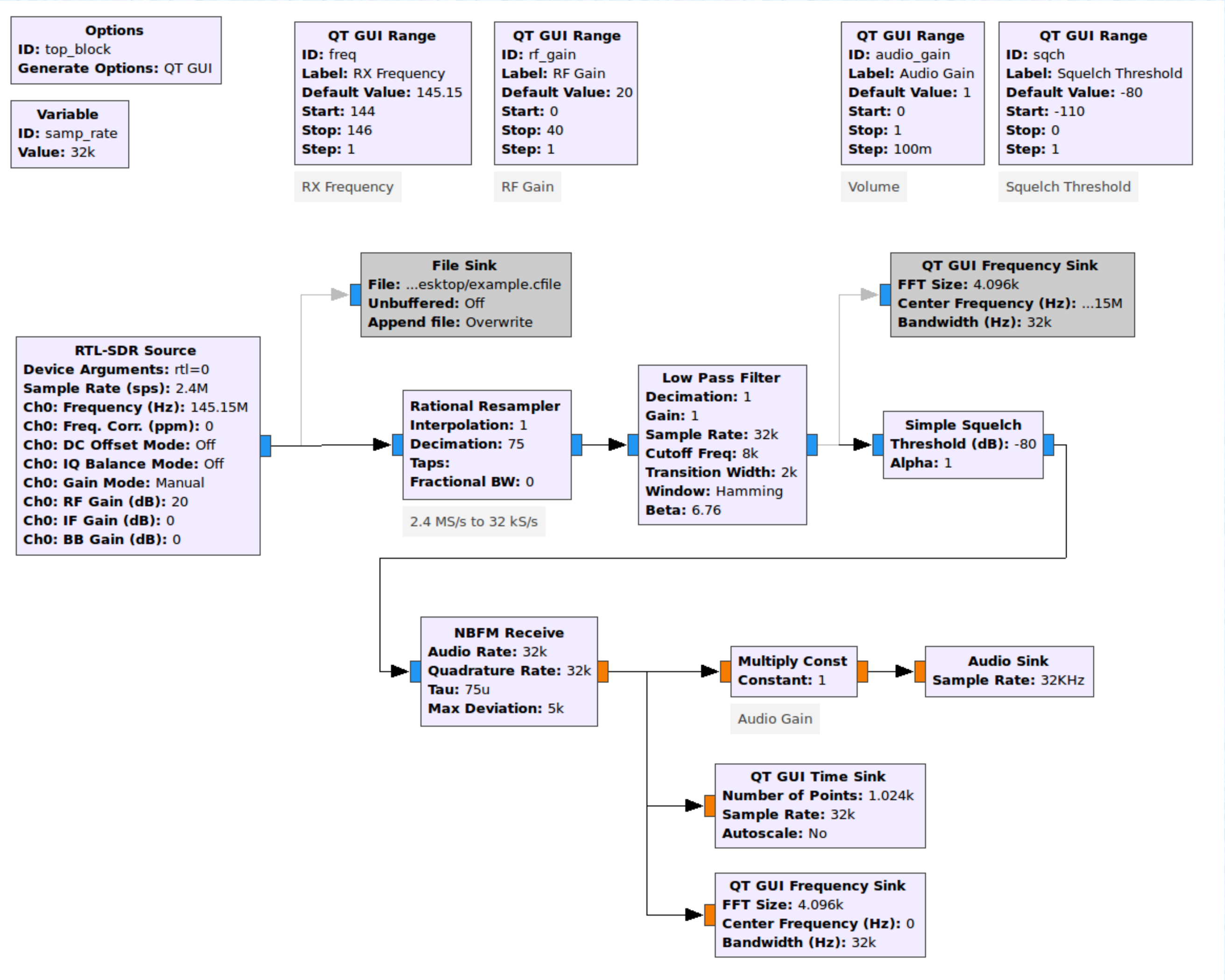
Bravo Alpha Oscar 2018

Data Types

- Data comes in different formats
- Most common is Complex samples in 32 bit floating point format
- Next most common is Real (not complex) samples in 32 bit floats
- “Help > Types” for info



Narrowband FM Receiver



Narrowband FM Receiver - Notes

- Soundcards will support different rates, 32 and 44.1 kHz pretty universal
- Thoughtful selection of SDR sampling rate makes decimation simple (1/75)
  - Avoid large fractions (i.e. 1023/127) as they require LOTs of computation
- Squelch is in dB Full Scale, not dBm or dbW
  - GNU Radio has no way of knowing an absolute power level

Narrowband FM Receiver - Notes

- NBFM block
  - Can decimate, but usually set output and input sample rates to the same
  - Deviation and pre-emphasis (tau) are dependent on the transmitter, default values will work in most cases

Underruns

- Soundcards and transmitters are hard-realtime systems, you must supply enough data to keep them always running
  - Failing to do so will cause an “underrun”
  - In RF will produce gaps in the transmission and splatter
  - In Audio will produce gaps and clicks
- GNU Radio will print “U” for underruns with USRPs and “aU” for soundcards (audio Underrun)

The Two-Clock Problem

- SDR Transmitter or receiver has an internal reference oscillator, so does a soundcard
- If the two references are not **EXACTLY** the same there's a problem
  - Source (producer) frequency > Sink (consumer) means too many samples are available, will build up a backlog of data to handle
    - In to Out delay will increase (Audio will lag)
  - Source < Sink means not enough data is available, underruns will occur

Mitigating the Two-Clock Problem

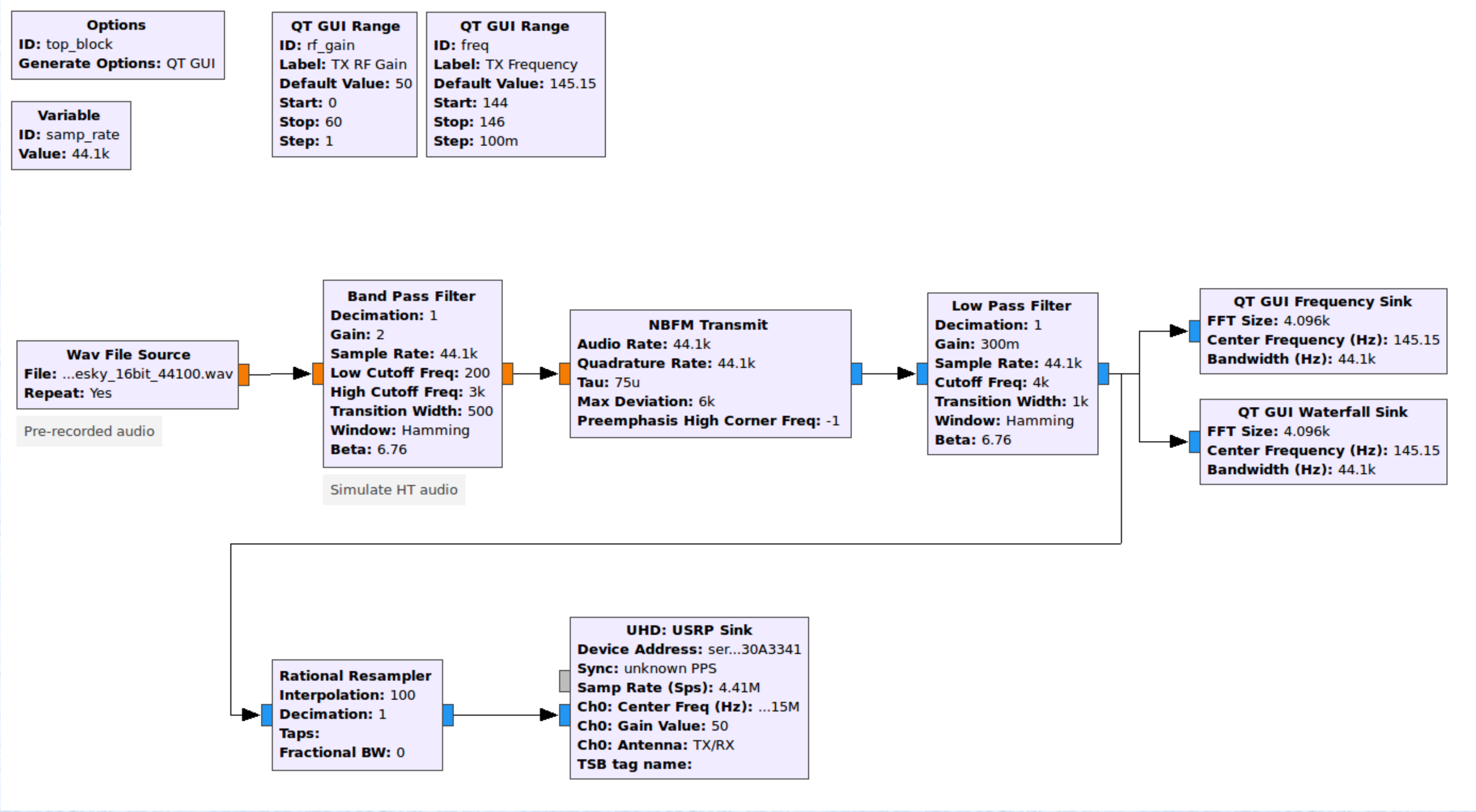
- Use the same reference oscillator for source and sink sample clocks (ADCs & DACs)
  - Great answer if using the same hardware for both, difficult (or impossible) with an SDR and soundcard
- Increase buffer sizes
  - Store more data before telling output to start
  - Reduces how often underruns occur
    - I.E. run out of data once a minute rather than 0.1 seconds

Mitigating the Two-Clock Problem

- In Linux (maybe OSX?):
  - Open the GNU Radio Config file
    - Click this icon 
    - Edit the path at the top to "~/.gnuradio"
    - Double click the config.conf file
  - OR type "gedit ~/.gnuradio/config.conf" into the terminal
  - Add the two highlighted lines:
    - This increases the soundcard buffer sizes

```
[audio_alsa]
default_input_device = default
default_output_device = default
nperiods = 16
period_time = 0.100
verbose = true
```

Narrowband FM Transmitter



Narrowband FM Transmitter - Notes

- USRP hardware sink sets transmit frequency, RF gain, and expected sample rate
  - USRP B200 (my demo hardware) is very flexible in sample rates, usually hardware will support specific rates
- Software interpolation/decimation will have sharper (better) filtering than FPGA or analog filters
  - This is a generalization but usually true
  - Interpolating by 100x means we have a clean signal but still very manageable sample rate (4.41 MS/s, easy for USB)

Narrowband FM Transmitter - Notes

- Use the time and frequency sinks to plot signals at different points (think spectrum analyzer and oscilloscopes when debugging)
- Confirm functionality off the air before including hardware (simulation)
- FM is forgiving with filtering
  - Accidentally generated 6 kHz deviation, filtered to 4k Hz, received with 5 kHz, still works
  - Partially thanks to filter transition bandwidth

Useful Tips

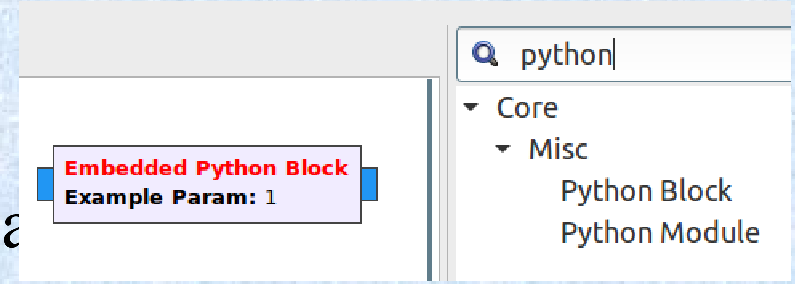
- Test/develop using a pre-recorded audio file
  - Expected format is 16 bit real valued samples
  - Sample rate chosen as 32 kHz to match what a soundcard (Mic in) would likely generate
- Add comments
  - Text box in the “Advanced” tab of each block
- Use variables and sliders (“Range” in QT”
  - Lets you experiment quickly with values to hand tune performance

Programming Languages

- GNU Radio has a core written in C++
  - The main engine and all default blocks are C++
- Python is wrapped around the C++
  - Generally considered more experimenter friendly
  - Only small performance hit as main work is done in C++ land
- GRC is entirely written in Python
  - But again, the engine is C++, so best of both worlds

Python Block

- Lets draw back the curtain and peek at the insides
- The "Embedded Python Block" lets you add custom code to a GRC flowgraph very easily
  - Code is stored in the .grc file
  - Default template supplies ba



Embedded Python Block

- Add a “Python Block” to the flowgraph, open it and click “Open in Editor” and use the Default
- The template has all the main features of a GNU Radio block setup already

```python
"""
Embedded Python Blocks:

Each time this file is saved, GRC will instantiate the first class it finds
to get ports and parameters of your block. The arguments to __init__  will
be the parameters. All of them are required to have default values!
"""

import numpy as np
from gnuradio import gr


class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self, example_param=1.0):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Embedded Python Block',   # will show up in GRC
            in_sig=[np.complex64],
            out_sig=[np.complex64]
        )
        # if an attribute with the same name as a parameter is found,
        # a callback is registered (properties work, too).
        self.example_param = example_param

    def work(self, input_items, output_items):
        """example: multiply with constant"""
        output_items[0][:] = input_items[0] * self.example_param
        return len(output_items[0])
```

Header and Includes

```
"""
Embedded Python Blocks:

Each time this file is saved, GRC will instantiate the first class it finds
to get ports and parameters of your block. The arguments to __init__  will
be the parameters. All of them are required to have default values!
"""

import numpy as np
from gnuradio import gr
```

- The red text surrounded by quotes is a comment explaining how the template works
- The import lines pull in code from gnuradio and numpy
  - numpy is a Python library of math functions that GNU Radio uses extensively
- You could add more imports to use other libraries

Class and Initialization

```
class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self, example_param=1.0):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Embedded Python Block',   # will show up in GRC
            in_sig=[np.complex64],
            out_sig=[np.complex64]
        )
```

- GNU Radio has several types (or “classes”) of blocks
  - We’re using a sync block since input and output rates are the same (synchronous)
- The next comment will appear in the block documentation tab
- The “__init__” function setups (initializes) our block
  - We have one parameter called example_param with a default value of 1.0

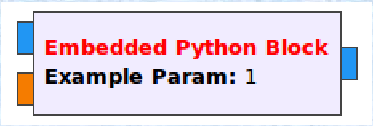
Block Initialization

```
class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self, example_param=1.0):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Embedded Python Block',   # will show up in GRC
            in_sig=[np.complex64],
            out_sig=[np.complex64]
        )
```

- GNU Radio already knows a lot about blocks. We just have to fill in the specific details by calling gr.sync_block.__init(….)
  - name is just for humans
  - in_sig/out_sig is the "signature" of the input/output
    - How many channels, what type of data (1 channel of complex data)
    - The data types are numpy since this is Python

Block Initialization - Continued

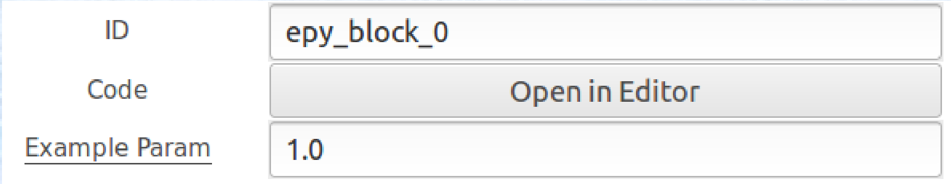
- in_sig=[np.complex64, np.float32] would be 1 channel complex and 1 channel real floats



- If you want to be able to change a value while the flowgraph is running (with a Range slider for instance) then create a "class attribute" like the following:

```
    # if an attribute with the same name as a parameter is found,
    # a callback is registered (properties work, too).
    self.example_param = example_param
```

- GRC will automatically add code to update the value correctly
  - Only values with an underline in GRC can be changed at runtime



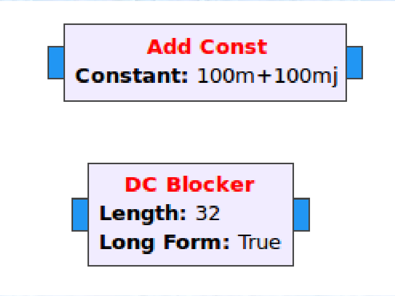
Doing *Work* on Samples

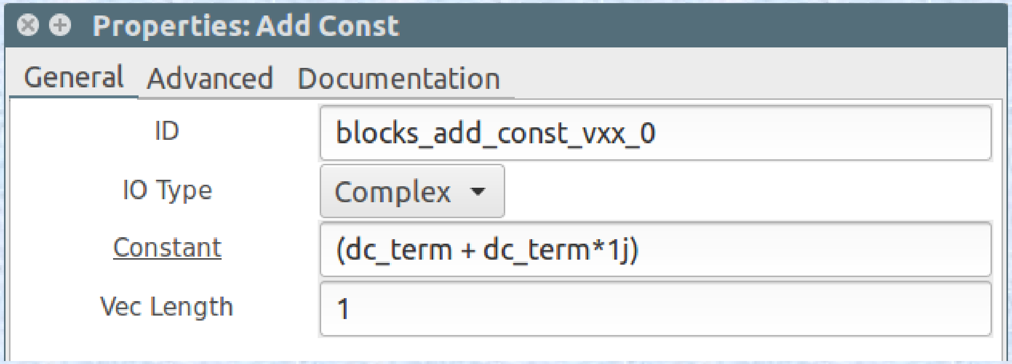
```
def work(self, input_items, output_items):
    """example: multiply with constant"""
    output_items[0][:] = input_items[0] * self.example_param
    return len(output_items[0])
```

- The main purpose of most blocks is to do something with or to samples
  - GNU Radio will call the *work* function with a bunch of input samples and a place to put the output samples
- The default template multiplies each sample by a value (example_param)
- We need to tell GNU Radio how many samples we've produced
  - In this case we've used all the input to make the same number of output samples
  - The *len* function gives the length of the output_items array, so we *return* that number to GNU Radio's engine
- Clearly some Python knowledge is needed, but most of the heavy lifting already done
- Great for implementing small pieces of math or functionality

DC Offset Example

- Same template but cleaned up
- Let's introduce a DC component to the signal
  - Usually a terrible idea
  - Could have used an *Add Const* block



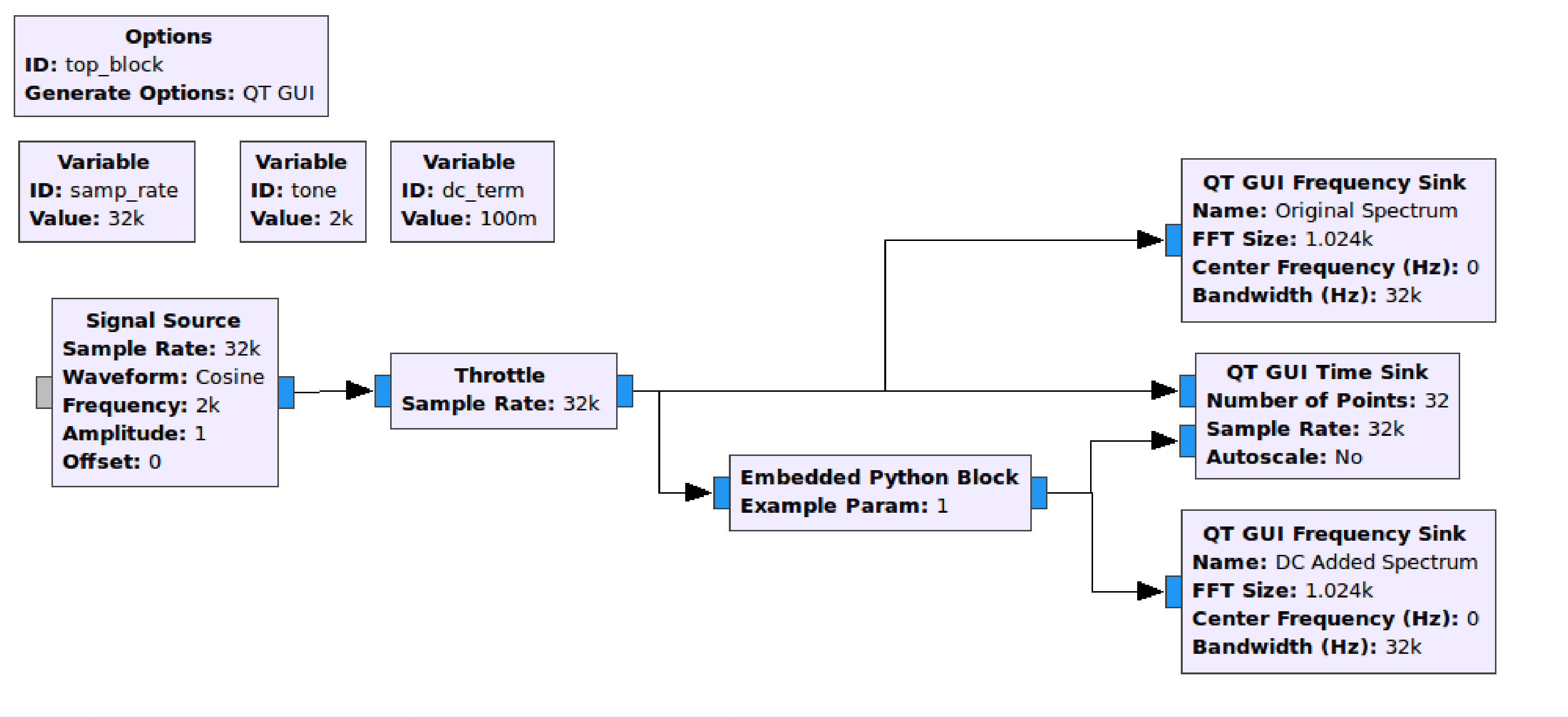


Trivia

Can remove a DC offset using the DC Blocker

DC Offset Test Setup

- Basic testing setup with an *Embedded Python Block*

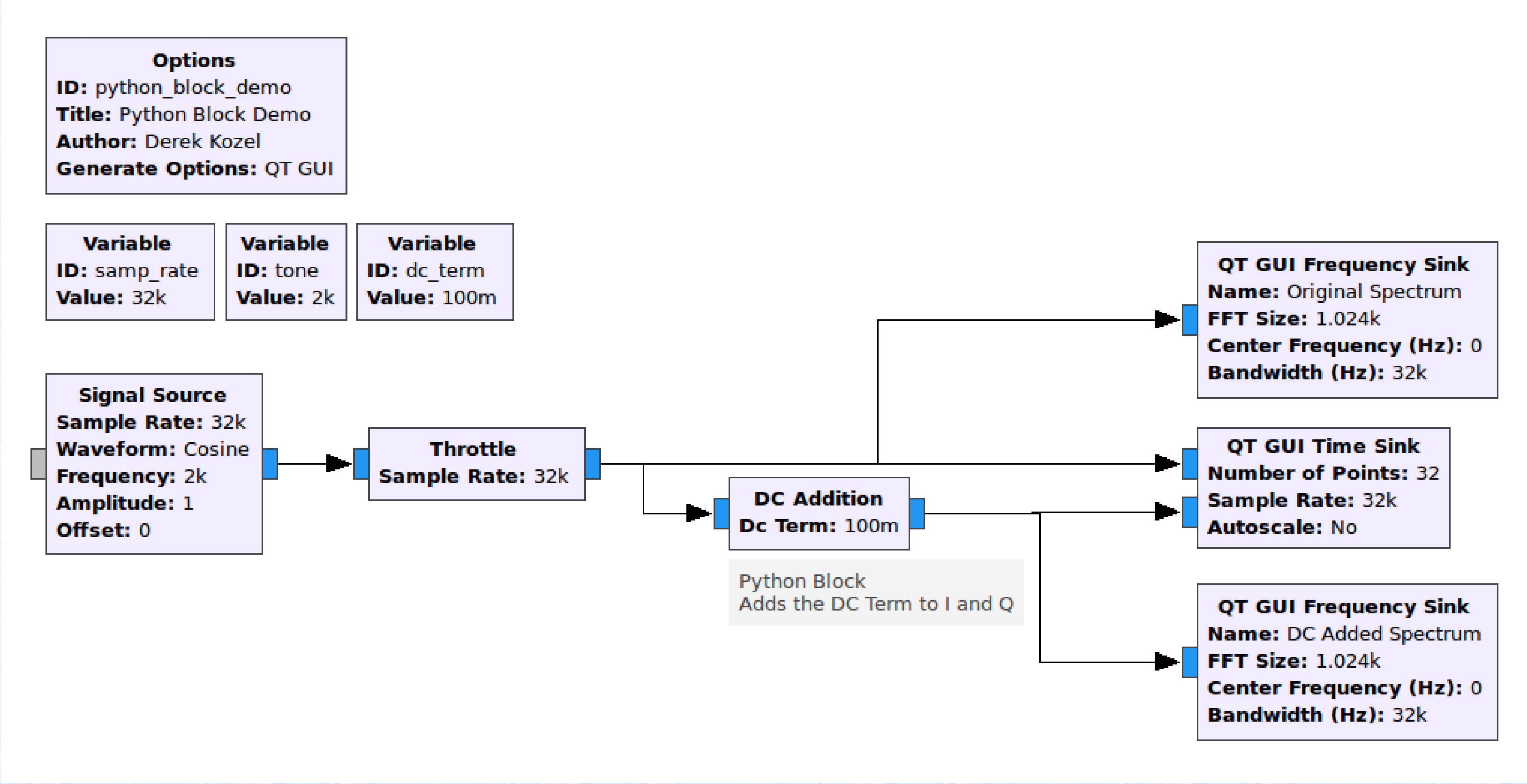


DC Offset Code

```
import numpy as np
from gnuradio import gr

class blk(gr.sync_block):
    # Block Documentation
    """DC Addition Block - Surely more is better!"""

    def __init__(self, dc_term=0.1):  # One parameter

        gr.sync_block.__init__(
            self,
            name='DC Addition',     # Will show up in GRC
            in_sig=[np.complex64], # Complex float 32 bit pairs
            out_sig=[np.complex64] # Complex float 32 bit pairs
        )

        self.dc_term = dc_term

    def work(self, input_items, output_items):

        # Add the value of "dc_term" to the I and Q parts of the signal
        # For example: output = input + (0.1 + j0.1)
        output_items[0][:] = input_items[0] + np.complex64(self.dc_term+self.dc_term*1j)

        # Tell GNU Radio's scheduler how many samples we are outputting
        return len(output_items[0])
```

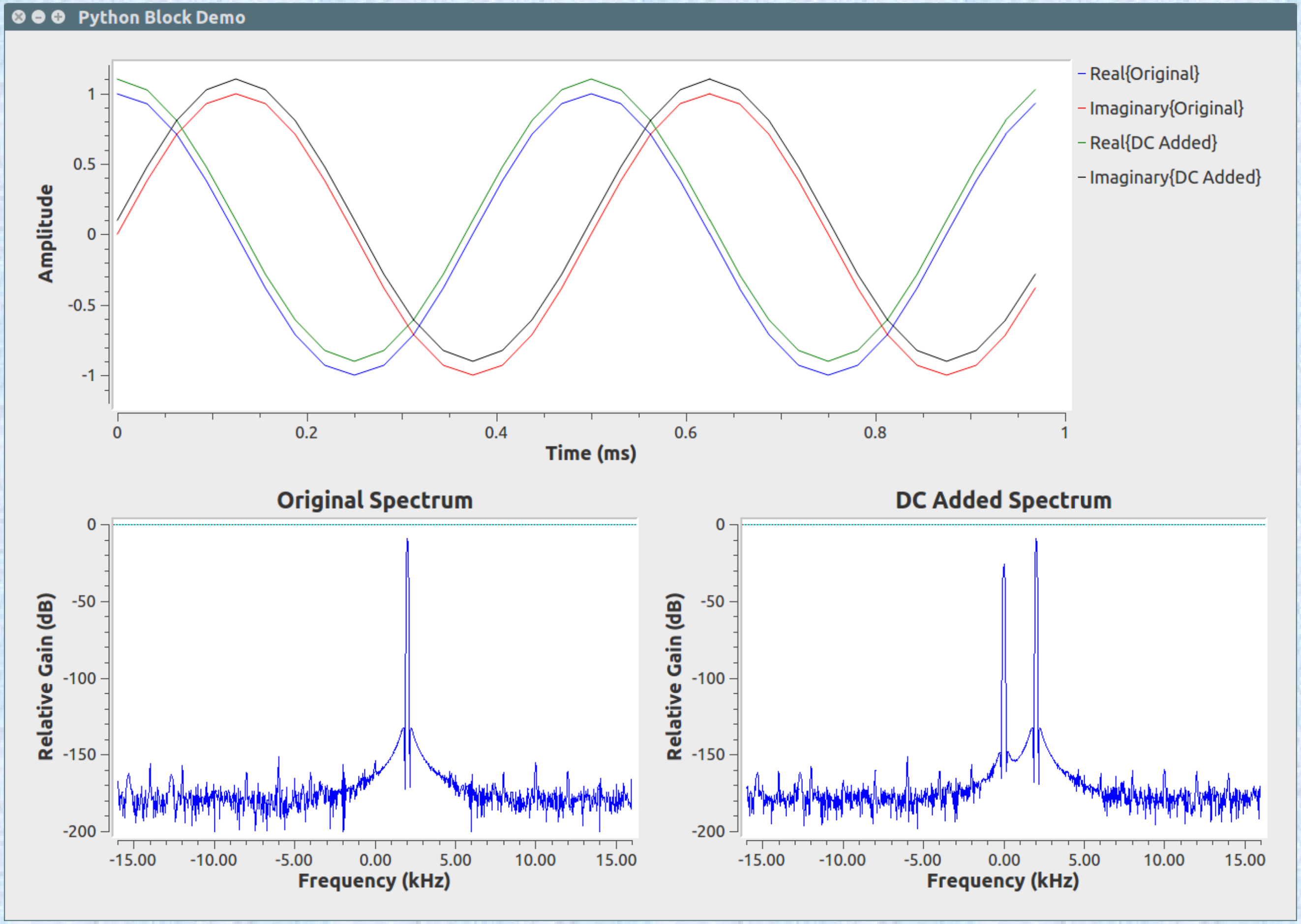
DC Offset Results

- Now looks like a real block



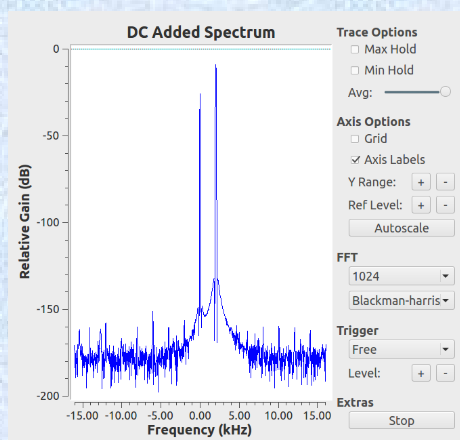
DC Offset Results

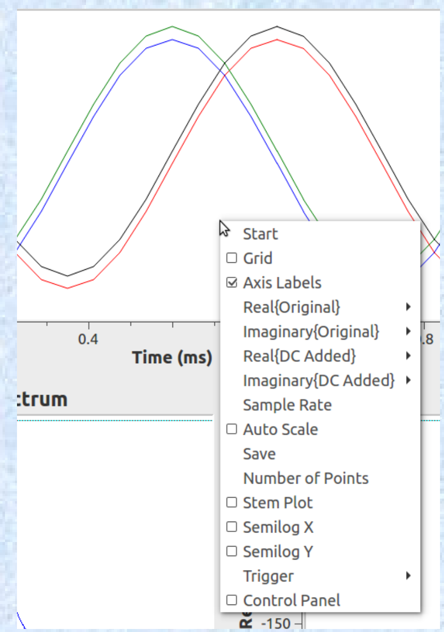
- DC offset clearly visible in time and frequency



Quick Tips

- Click on the line labels in the Time plot to hide or show a particular line
  - Works on other visual sinks too
- Middle mouse click on a QT plot to bring up a menu of options.
- Enable a Control Panel in the Advanced Tab





User Manual & Documentation

- A bit spread out and wanting in depth in spots
- User Manual: www.gnuradio.org/doc/doxygen
  - Generated from the C++
  - Useful for finding out more about blocks
  - Talks about the design of the core engine and code
- Python Manual: www.gnuradio.org/doc/sphinx
  - Generated from the Python
  - Does **not** cover many of the topics in main manual
  - Likely to be combined with the C++ in the next year

User Manual & Documentation

- Wiki: http://wiki.gnuradio.org
  - Several sets of tutorials
  - Presentations from other classes and events
  - Working groups and developer info
  - GNU Radio Conference info
    - Links to videos and slides from the talks
  - Lots of outdated pages, getting cleaner over time

Main Website

- www.gnuradio.org
- Blog
  - Short and long posts about significant events
- Releases
  - Description of changes in new versions
- Links to everything on the previous page

Mailing List

- discuss-gnuradio
  - Email list for general discussion
  - Lots of helpful people
    - Explain your question or problem clearly and you’re almost certain to get quick and useful responses
  - Announcements about releases and OOT development from other users and companies
  - Decent Amateur Radio presence already

Slack

- Real time text chat room
  - Lots of people hanging around willing to chat about most technical things
  - Some true beacons of knowledge about GNU Radio, DSP, SDR, etc
- Sign up at http://slack.gnuradio.org
  - Automatically sends you an invite
- Log in at http://gnuradio.slack.org
- If you prefer IRC it is linked to #gnuradio on irc.freenode.com

Community Events

- Upcoming Developer Hackfest
  - Based in California, but coordinated online with groups in the UK, Germany, and around the world
- FOSDEM – Belgium, February 2/3
  - 8,000+ software developers
  - Free Software Radio room with a full day of talks
  - Strong Amateur Radio presence

Community Events

- SDR Academy- Friedrichshafen
  - Same time and place as HAMRADIO
  - Full day of talks about SDR
    - GNU Radio usually has one or two
- French GNU Radio Days
  - Ran in 2018 for the first time
  - Back again this year
  - Two days of talks and tutorials
  - Small but hopefully growing

GNU Radio Conference

- Run each year by the GNU Radio Foundation
- 5 days of talks, tutorials, and workshops
  - Talks recorded and slides available online
  - https://www.youtube.com/channel/UCceoapZVEDCQ4s8y16M7Fng
- Historically in the USA, strong interest in a European version soon

Next Steps

- Do the Guided Tutorials
- Receive some signals over the air at home
- Ask a question on the mailing list or Slack
- Try running GQRX (already on the Live USB)
- Watch a GNU Radio Conference talk
  - Suggestion: https://www.youtube.com/watch?v=yT1DFxDgI_8

Wrapping Up

- Thank you for taking the class!
- Let me/us know any questions

derek@bitstovolts.com

Twitter: *@derekkozel*

Credits and Info

- Many thanks to the RSGB Legacy Fund and UK Microwave Group
- John Worsnop – G4BAO
- Dr Heather Lomond – M0HMO

- All slides are licenced as

Creative Commons Attribution-ShareAlike 4.0 International